MIPS Processor

Fully Pipelined Architecture

Table of contents

List of figures

Introduction

IN this documentation, we delve into MIPS (Microprocessor without Interlocked Pipeline Stages) architecture exploring the efficient instructions for sequential and parallel execution and the essential concept of pipeline architecture. Moreover, In the conclusion of the document, you'll find a description of our project's design and test bench, created using the VIVADO tool and Questa sim.

MIPS is a family of reduced instruction set computer (RISC) architectures developed by MIPS Computer Systems, MIPS stands for (Microprocessor without Interlocked Pipeline Stages) or (million instructions per second),now MIPS Technologies, based in the United States. MIPS processors are known for their simplicity and efficiency, with a fixed-length instruction format and a focus on a streamlined instruction set. It serves as a comprehensive guide to understanding MIPS processors and their intricate pipeline design.

Understanding pipeline architecture is essential for taking the inner workings of modern processors. Pipelining, a technique to enhance instruction throughput, breaks down the instruction execution process into multiple stages. In the context of MIPS architecture, comprehending the pipeline structure provides insights into the sequential behavior of fetching, decoding, and executing instructions. However, it introduces challenges related to hazards and dependencies that need to be addressed for optimal performance. Knowledge of these concepts is essential for computer architects, engineers, and programmers working with MIPS processors to design efficient and high-performance systems, and as we know after implementing the pipeline in the MIPS processor architecture, the processor executes an instruction every clock cycle. This significant improvement in throughput is a direct result of the pipelining technique, The MIPS pipeline typically consists of five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). Each stage is responsible for a specific operation in the instruction execution process, in a pipelined MIPS processor, multiple instructions are simultaneously in different stages of execution. As soon as one instruction completes a stage, it advances to the next stage, allowing the next instruction to enter the pipeline. This overlapping of instruction execution stages enables the processor to handle multiple instructions concurrently, thereby improving throughput.

Section 1: The architecture

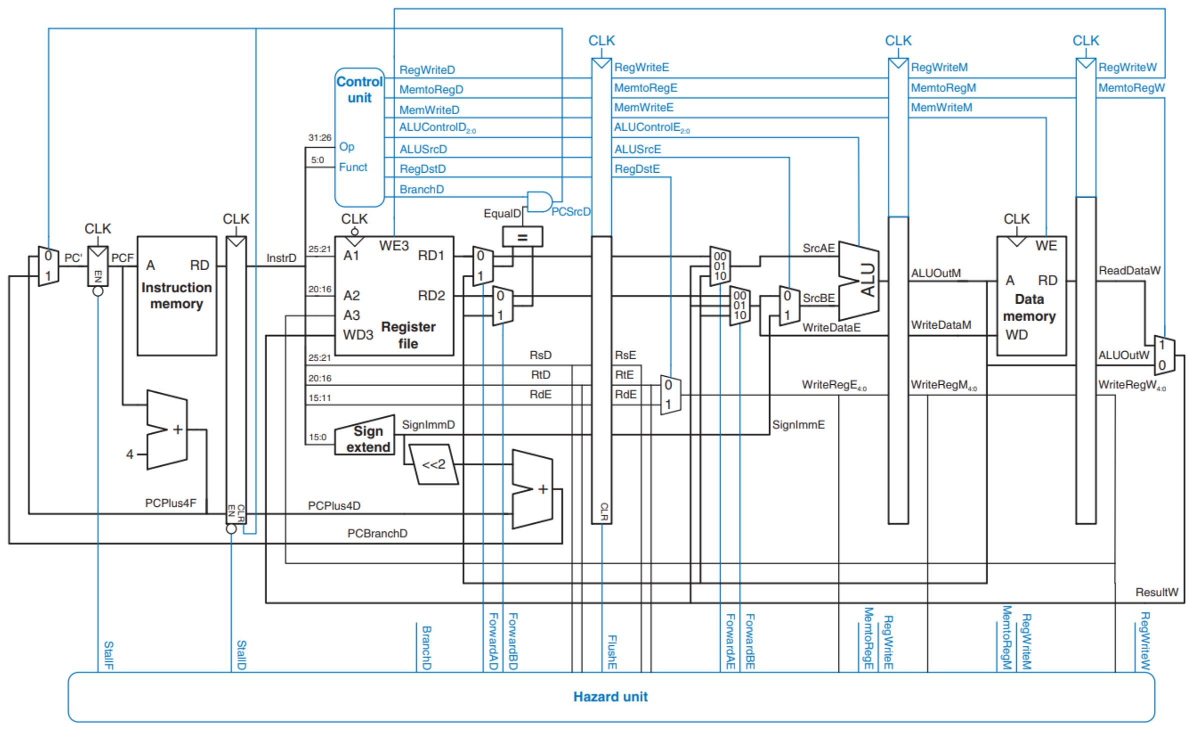


Figure 1 : MIPS Architecture

1.1 Pipelined Processor

Pipelining is a powerful way to improve the throughput of a digital system. We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can be executed simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is almost five times faster. Hence, the latency of each instruction is ideally unchanged, but the throughput is ideally five times better.

Pipelining introduces some overhead (due to potential occurrence of pipeline hazards and the addition of more logic components), so the through put will not be quite as high as we might ideally desire but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined.

1.2 Instructions

In MIPS architecture, instructions are categorized into distinct types, each serving specific computational or control flow purposes.

3.1 R-Type Instructions

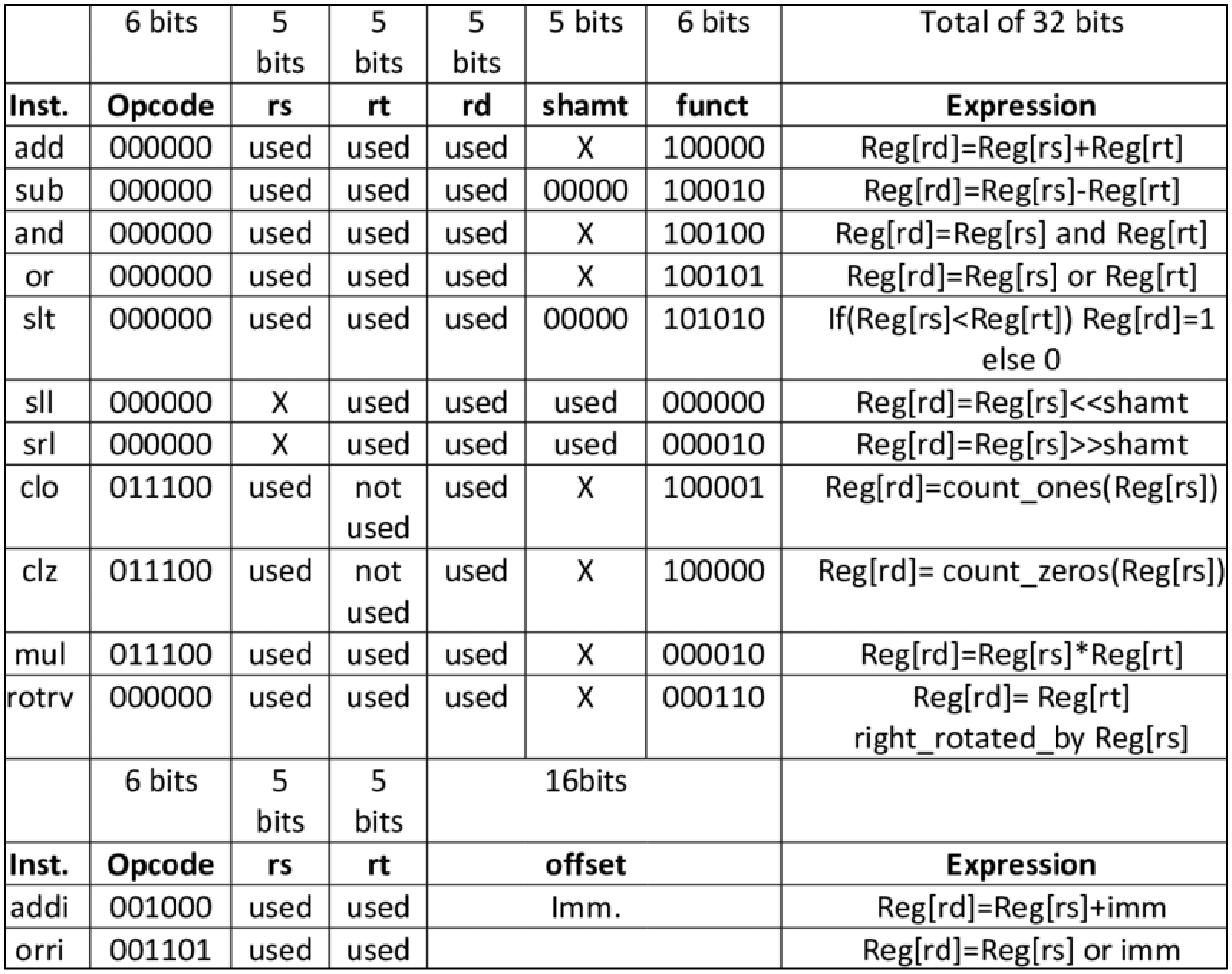
R-Type instructions are characterized by operations performed on data stored within registers. These instructions format consists of an opcode, source registers (rs and rt), a destination register (rd), a shift amount which decide the amount of shift you need (shamt), and a function code (funct Which decides what operation will be performed. In R-Type instructions, we implemented fundamental arithmetic and logical operations, including addition, subtraction, bitwise AND, bitwise OR, and set less than.

3.2 I-Type Instructions

I-Type instructions combine register-based operations with immediate values. They are utilized for operations involving data from registers as well as immediate constants or offsets. The format consists of an opcode, source register (rs), destination register (rt), and an immediate value. These instructions are extensively used for tasks such as loading data from memory (lw), storing data to memory (sw), and conditional branching (beq).

3.3 J-Type Instructions

J-Type instructions primarily facilitate control flow operations, particularly unconditional branching. They involve specifying a target address for the jump. These instructions usually consist of an opcode and a memory address. Examples of J-Type instructions include j for unconditional jump.



	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Total of 32 bits
Inst.	**Opcode**	**rs**	**rt**	**rd**	**shamt**	**funct**	**Expression**
add	000000	used	used	used	X	100000	Reg[rd]=Reg[rs]+Reg[rt]
sub	000000	used	used	used	00000	100010	Reg[rd]=Reg[rs]-Reg[rt]
and	000000	used	used	used	X	100100	Reg[rd]=Reg[rs] and Reg[rt]
or	000000	used	used	used	X	100101	Reg[rd]=Reg[rs] or Reg[rt]
slt	000000	used	used	used	00000	101010	If(Reg[rs]<Reg[rt]) Reg[rd]=1 else 0
sll	000000	X	used	used	used	000000	Reg[rd]=Reg[rs]<<shamt
srl	000000	X	used	used	used	000010	Reg[rd]=Reg[rs]>>shamt
clo	011100	used	not used	used	X	100001	Reg[rd]=count_ones(Reg[rs])
clz	011100	used	not used	used	X	100000	Reg[rd]= count_zeros(Reg[rs])
mul	011100	used	used	used	X	000010	Reg[rd]=Reg[rs]*Reg[rt]
rotrv	000000	used	used	used	X	000110	Reg[rd]= Reg[rt] right_rotated_by Reg[rs]
	6 bits	5 bits	5 bits	16bits			
Inst.	**Opcode**	**rs**	**rt**	**offset**			**Expression**
addi	001000	used	used	Imm.			Reg[rd]=Reg[rs]+imm
orri	001101	used	used				Reg[rd]=Reg[rs] or imm

Figure 2 : MIPS instruction set.

1.3 Hazards

While the MIPS pipelined processor offers numerous advantages, it also presents several challenges that require careful management.

3.1 Structural Hazards

Structural hazards occur when the hardware resources required to execute instructions are not available simultaneously.

3.2 Data Hazards

Data hazards arise due to dependencies between instructions, impacting the flow of data through the pipeline. The main types of data hazards in MIPS pipelines are : Read-after-write (RAW) hazards: Occur when an instruction tries to read data from a register that has been modified by a preceding instruction in the pipeline but has not yet been written back to the register file.

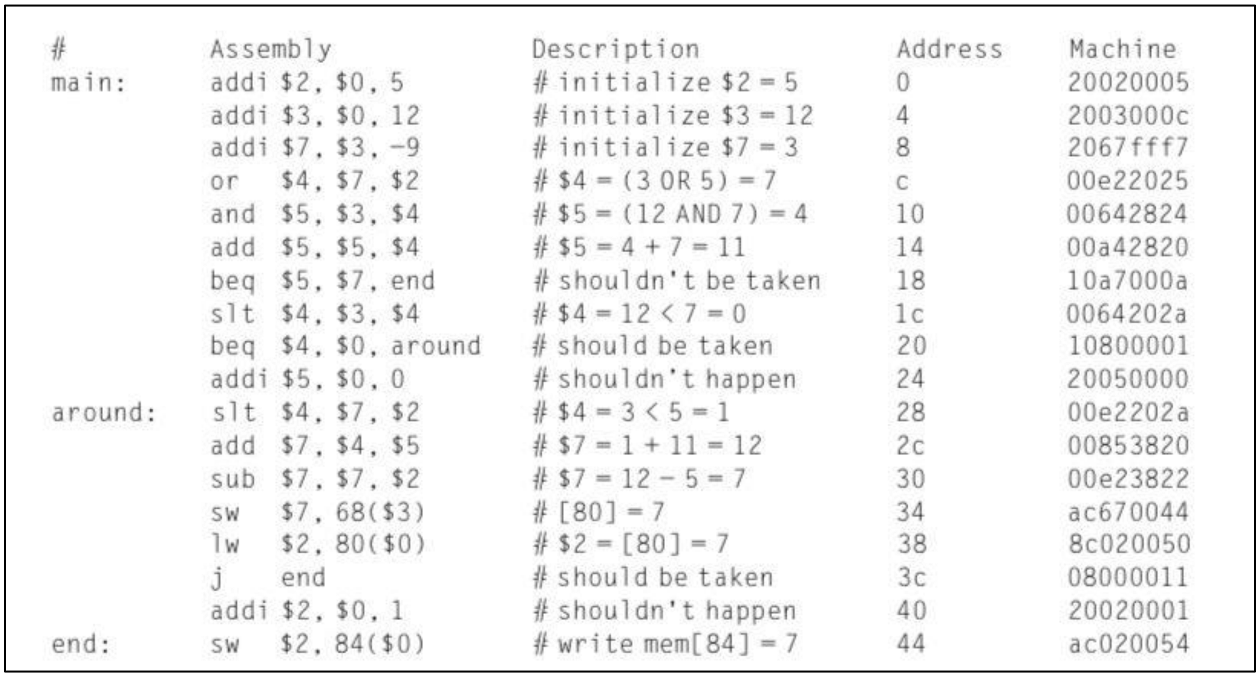
Write-after-read (WAR) hazards: Arise when an instruction writes to a register that a subsequent instruction tries to read from before the write operation is completed. Write-after-write (WAW) hazards: Manifest when two instructions attempt to write to the same register in consecutive pipeline stages, potentially causing unpredictable behavior. Read-after-read(RAR): It doesn't consider a hazard.

3.3 Control Hazards

Control hazards occur when the pipeline encounters branches or jumps, leading to uncertainty about the next instruction to execute a Hazard Unit which has signals Forward ,load stall and branch stall signals to manage these challenges. Forward signals directly pass results to dependent instructions, mitigating data hazards. Load Stall signals tackle pipeline stalls from load instruction dependencies, ensuring synchronization. Branch Stall signals assist in controlling uncertainties from branch instructions.

Section 3 : Results

We have use a Testbench that encompasses nearly all instructions as shown in figure 6. We achieved this by loading a program into the instruction memory and analyzing the expected file output. Our objective is to verify that the code yields the value 7 at memory address 80 and at address 84.



#	Assembly	Description	Address	Machine
main:	addi $2, $0, 5	# initialize $2 = 5	0	20020005
	addi $3, $0, 12	# initialize $3 = 12	4	2003000c
	addi $7, $3, −9	# initialize $7 = 3	8	2067fff7
	or $4, $7, $2	# $4 = (3 OR 5) = 7	c	00e22025
	and $5, $3, $4	# $5 = (12 AND 7) = 4	10	00642824
	add $5, $5, $4	# $5 = 4 + 7 = 11	14	00a42820
	beq $5, $7, end	# shouldn't be taken	18	10a7000a
	slt $4, $3, $4	# $4 = 12 < 7 = 0	1c	0064202a
	beq $4, $0, around	# should be taken	20	10800001
	addi $5, $0, 0	# shouldn't happen	24	20050000
around:	slt $4, $7, $2	# $4 = 3 < 5 = 1	28	00e2202a
	add $7, $4, $5	# $7 = 1 + 11 = 12	2c	00853820
	sub $7, $7, $2	# $7 = 12 − 5 = 7	30	00e23822
	sw $7, 68($3)	# [80] = 7	34	ac670044
	lw $2, 80($0)	# $2 = [80] = 7	38	8c020050
	j end	# should be taken	3c	08000011
	addi $2, $0, 1	# shouldn't happen	40	20020001
end:	sw $2, 84($0)	# write mem[84] = 7	44	ac020054

Figure 3 : Assembly code for Testbench.

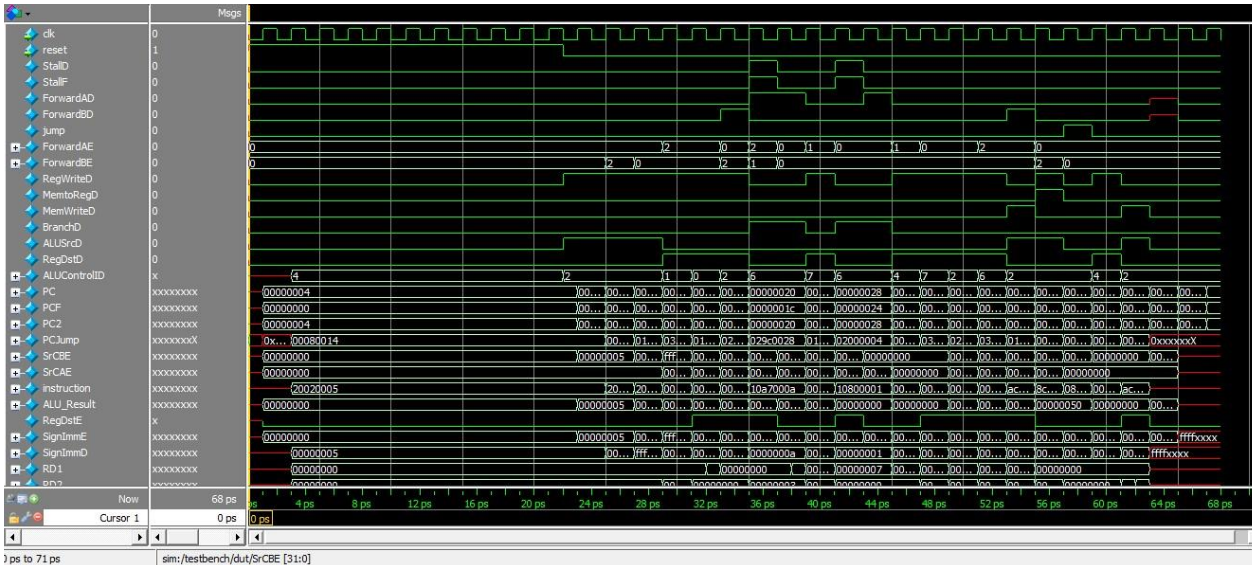
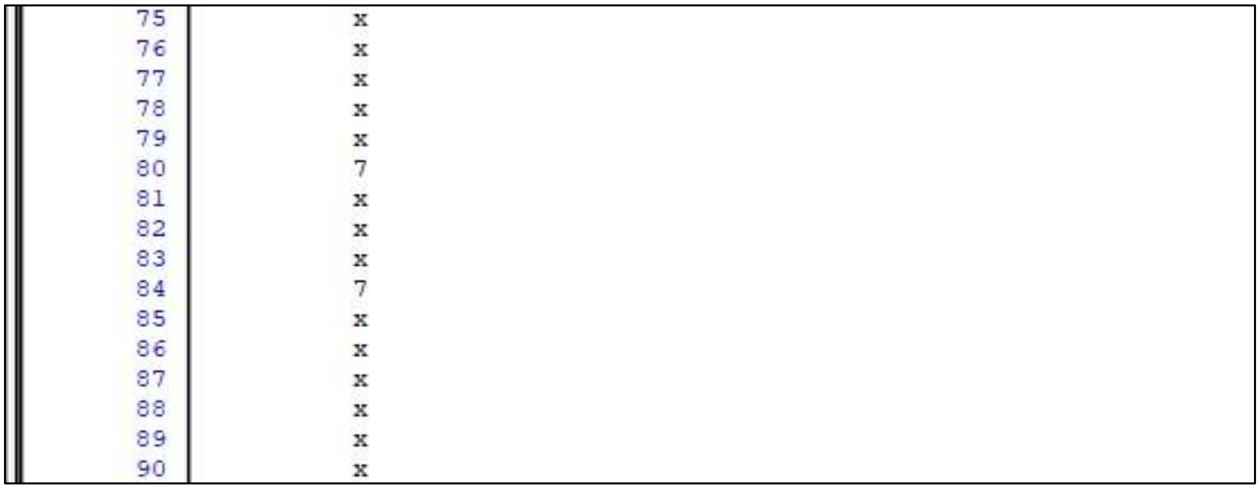


Figure 4 : Output waveform.



75	x
76	x
77	x
78	x
79	x
80	7
81	x
82	x
83	x
84	7
85	x
86	x
87	x
88	x
89	x
90	x

Figure 5 : Final Memory content.

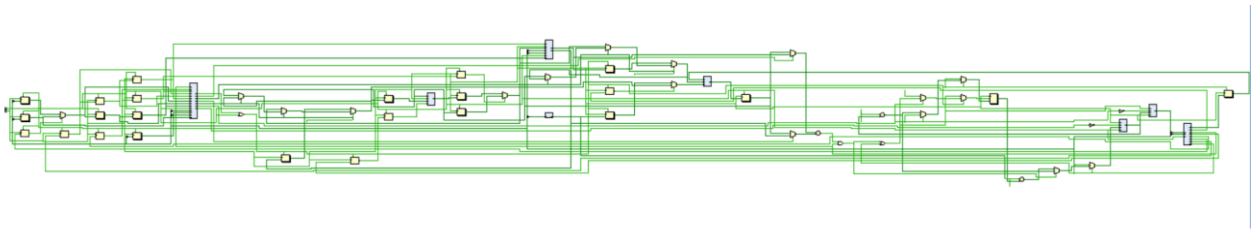


Figure 6 : Elaborate design .

References

1. **Harris, David, and Sarah Harris. Digital Design and Computer Architecture. Morgan Kaufmann, 2012.**